Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

TRITA-EECS-AVL-2020:4
ISBN 100-

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik i .

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***Superoptimization of WebAssembly Bytecode***
   Javier Cabrera-Arteaga,Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
   Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
   `https://doi.org/10.1145/3397537.3397567`

2. ***CROW: Code Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
   Network and Distributed System Security Symposium (NDSS 2021), MADWeb
   `https://doi.org/10.14722/madweb.2021.23004`

3. ***Multi-Variant Execution at the Edge***
   Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
   Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
   `https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

4. ***WebAssembly Diversification for Malware Evasion***
   Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
   Computers & Security, Volume 131, 2023
   `https://www.sciencedirect.com/science/article/pii/S0167404823002067`

5. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
   Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry

6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
   Javier Cabrera-Arteaga,Martin Monperrus, Benoit Baudry
   11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
   `https://doi.org/10.1145/3358504.3361228`

ACKNOWLEDGEMENT

List of commonly used acronyms:

Wasm WebAssembly

Contents

Part I

Thesis

01 INTRODUCTION

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.1 Background

TODO Motivate with the open challenges.

1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.3 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: Mapping of the contributions to the research papers appended to this thesis.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

1.5 Summary of research papers

P1: Superoptimization of WebAssembly Bytecode.

P2: CROW: Code randomization for WebAssembly bytecode.

P3: Multivariant execution at the Edge.

P4: Wasm-mutate: Fast and efficient software diversification for WebAssembly.

P5: WebAssembly Diversification for Malware evasion.

1.6 Thesis outline

02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

TODO The concept of managed and unmanaged data in Wasm from Lehman.

2.1.1 WebAssembly toolchains

TODO Mention, stress the landscape of tools that involve Wasm. Include analysis tools, fuzzers, optimizers and malware detectors.
TODO End up motivating the need of Software Diversification for: testing and reliability.

2.2 Software diversification

2.3 Generating Software Diversification

2.3.1 Variants generation

2.3.2 Variants equivalence

2.4 Exploiting Software Diversification

2.4.1 Defensive Diversification

2.4.2 Offensive Diversification

03 AUTOMATIC SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

WebAssembly programs are produced ahead of time through a process that begins with the source code and moves through the compiler, ultimately resulting in a WebAssembly program. Software Diversification can be achieved at any of these stages. Diversifying at the source code stage, however, is not practical due to the necessity of creating a distinct diversifier for each language compatible with WebAssembly. In contrast, focusing on the compiler stage presents a viable option, especially considering that 70% of WebAssembly binaries are created using LLVM-based compilers, as noted by Hilbig et al. [**?**]. Furthermore, implementing diversification at the WebAssembly program stage stands as the most generic strategy, applicable to any WebAssembly program in the wild. Therefore, this thesis focuses to the exploration of diversification strategies at the compiler and WebAssembly program stages, employing two main approaches: compiler-based and binary-based.

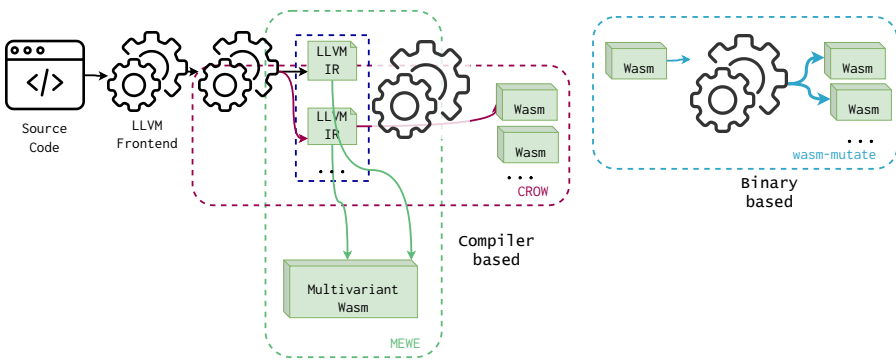

Figure 3.1: Approach landscape.

Our compiler-based strategies are depicted in red and green in Figure 3.1.

This approach introduces a diversifier component in the LLVM pipeline, generating LLVM IR variants and producing artificial software diversity for Wasm. This strategy encompasses two tools: CROW [**?**], which creates Wasm program variants, and MEWE [**?**], which merges these variants to foster multivariant execution for Wasm. In contrast, the binary-based strategy, illustrated in blue in Figure 3.1, offers diversification for any WebAssembly program. wasm-mutate [**?**] generates a pool of WebAssembly program variants through rewriting rules upon an e-graph [**?**] data structure, eliminating the need for compiler tuning. This dissertation contributes to the field of Software Diversification for WebAssembly, presenting three main technical contributions: CROW, MEWE, and wasm-mutate, which will be elaborated upon in the subsequent sections.

3.1 CROW: Code Randomization of WebAssembly

This section details CROW [**?**], represented as the red squared tooling in Figure 3.1. CROW is designed to produce semantically equivalent Wasm variants from the output of an LLVM front-end, utilizing a custom Wasm LLVM backend to craft Wasm binary variants.

Figure 3.2 illustrates CROW's workflow in generating program variants, a process compound of two core stages: *exploration* and *combining*. During the *exploration* stage, CROW processes every instruction within each function of the LLVM input, creating a set of functionally equivalent code variants. This process ensures a rich pool of options for the subsequent stage. In the *combining* stage, these alternatives are assembled to form diverse LLVM IR variants, a task achieved through the exhaustive traversal of the power set of all potential combinations of code replacements. The final step involves the custom Wasm LLVM backend, which compiles the crafted LLVM IR variants into Wasm binaries.

3.1.1 Variants' generation

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On

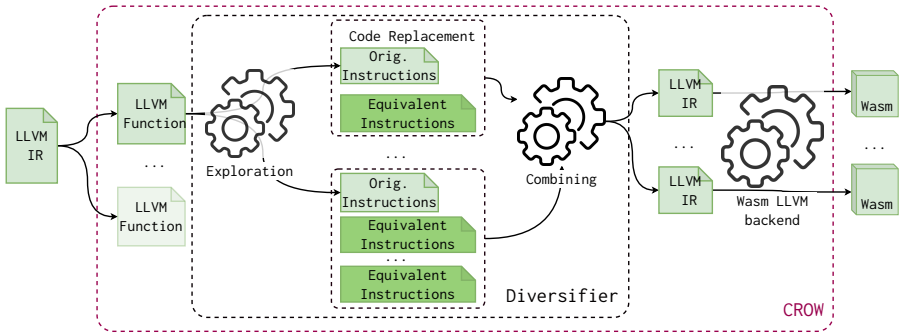

Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We modify Souper [**?**] to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until a timeout is reached or the replacements grow to a size larger that a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of the commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

3.1.2 Constant inferring

One of the code transformation strategies of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1     int babbage() {
2         int current = 0,
3             square;
4         while ((square=current*current) %
              ↪ 1000000 != 269696) {
5             current++;
6         }
7         printf ("The number is %d\n",
              ↪ current);
8         return 0 ;
9     }
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current);
    return 0 ;
}
```

both statically and at runtime[1].

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the Wasm backend. We disable all optimizations in the Wasm backend that could reverse the CROW transformations. In the following enumeration, we list three concrete optimizations that we remove from the Wasm backend.[2]

- Constant folding: this optimization calculates the operation over two (or more) constants in compiling time, and replaces the original expression by its constant result. For example, let us suppose $a = 10 + 12$ a subexpression to be compiled, with the original optimization, the Wasm backend replaces it by $a = 22$.
- Expressions normalization: in this case, the comparison operations are normalized to its complementary operation, e.g. $a > b$ is always replaced by $b <= a$.
- Redundant operation removal: expressions such as the multiplication of variables by $a = b2^n$ are replaced by shift left operations $a = b << n$.

■ 3.1.3 CROW instantiation

Let us illustrate how CROW works with the example code in Listing 3.3. The `f` function calculates the value of $2 * x + x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 3.4. CROW potentially finds two integer returning instructions to look for variants, as the right-most part of Listing 3.4 shows.

Listing 3.3: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

CROW, detects `code_1` and `code_2` as the enclosing boxes in the left most part of Listing 3.4 shows. CROW synthesizes $2 + 1$ candidate code replacements for each code respectively as the green highlighted lines show in the right most parts of Listing 3.4. The baseline strategy of CROW is

[1]Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in **??**.

[2]We only illustrate three of the removed optimization for the sake of simplicity.

Listing 3.4: LLVM's intermediate representation program, its extracted instructions and replacement candidates. Gray highlighted lines represent original code, green for code replacements.

```
define i32 @f(i32) {

2code 211.5103.5cm
1code 111.53.53.0cm
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

ret i32 %3
}

define i32 @main() {
%1 = tail call i32 @f(
     i32 10)
ret i32 %1
}
```

```
Replacement candidates for
      code_1
%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
      code_2
%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

Listing 3.5: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

%2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2

%2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2
```

```
%2 = mul nsw i32 %0,2
%3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0
%3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32
%3 = mul nsw %0, 3:i32
```

to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 3.5 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 3.6 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_2` by `mul nsw %0, 3`, turns `code_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 3.5 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon a *code replacement overlapping*.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to

Listing 3.6: Wasm program variants generated from program Listing 3.3.

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 2
   i32.mul
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 1
   i32.shl
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   local.get 0
   i32.add
   local.get 0
   i32.add
```

```
func $f (param i32) (result i32)
   local.get 0
   i32.const 3
   i32.mul
```

generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

■ 3.1.4 Combining replacements

When we retarget Souper, to create variants, we recombine all code replacements, including those for which a constant inferring was performed. This allows us to create variants that are also better than the original program in terms of size and performance. Most of the Artificial Software Diversification works generate variants that are as performant or iller than the original program. By using a superdiversifier, we could be able to generate variants that are better, in terms of performance, than the original program. This will give the option to developers to decide between performance and diversification without sacrificing the former.

On the other hand, when Souper finds a replacement, it is applied to all equal instructions in the original LLVM binary. In our implementation, we apply the transformation only to the instruction for which it was found in the first place. For example, if we find a replacement that is suitable for N difference places in the original program, we generate N different programs by applying the transformation in only one place at a time. Notice that this strategy provides a combinatorial explosion of program variants as soon as the number of replacements increases.

3.2 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE). The tool generates application-level multivariant binaries without changing the operating system or Wasm runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as the green squqred tooling in Figure 3.1 shows. MEWE inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE from the blue highlighted square in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, allowing the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant Wasm binary using the same custom Wasm LLVM backend from CROW. Once generated, the multivariant Wasm binary can be deployed to any Wasm engine.

3.2.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

In Figure 3.4, we show the original static call graph for an original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The gray nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each

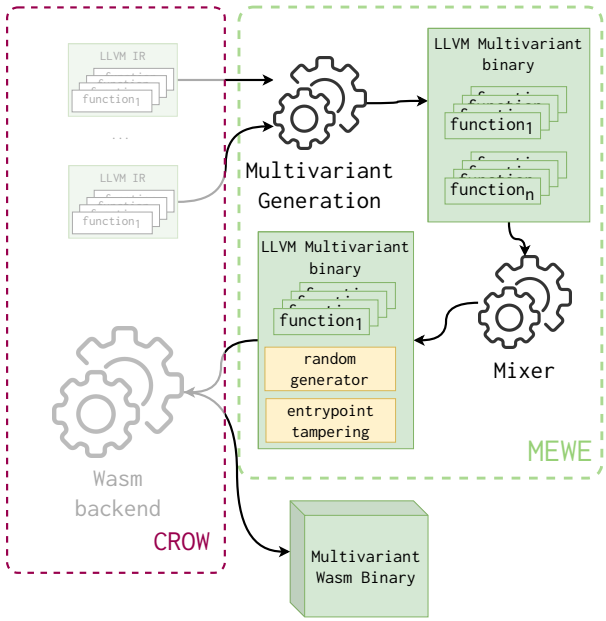

Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a Wasm multivariant binary ready to be deployed.

dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In Listing 3.7, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again.

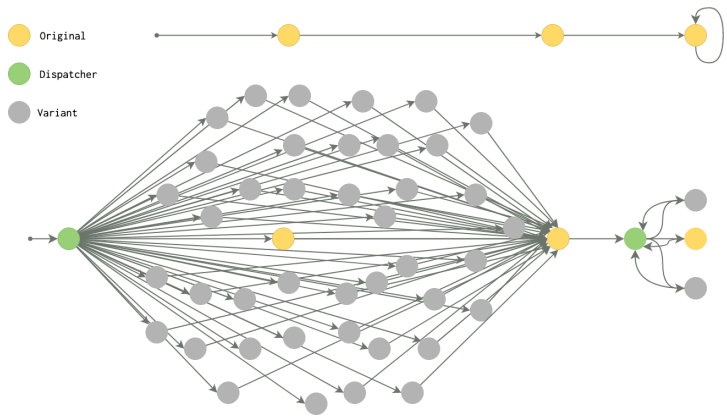

Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in gray), dispatchers (in green), and original functions (in yellow).

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.7: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

3.2.2 MEWE's Mixer

TODO Augment the description of the MIXER.

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant Wasm binary. We use the Rustc compiler[3] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

TODO Blend this to the need of wasm-mutate

3.3 Wasm-mutate

In this section we present wasm-mutate, a tool to diversify WebAssembly binaries and produce semantically equivalent variants. wasm-mutate is highlighted as the blue squared tooling in Figure 3.1. The primary objective of wasm-mutate is to generate semantically equivalent variants from a given WebAssembly binary input. wasm-mutate's central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules. It leverages a comprehensive set of rewrite rules, boosted by a diversification space traversals using e-graphs [**?**].

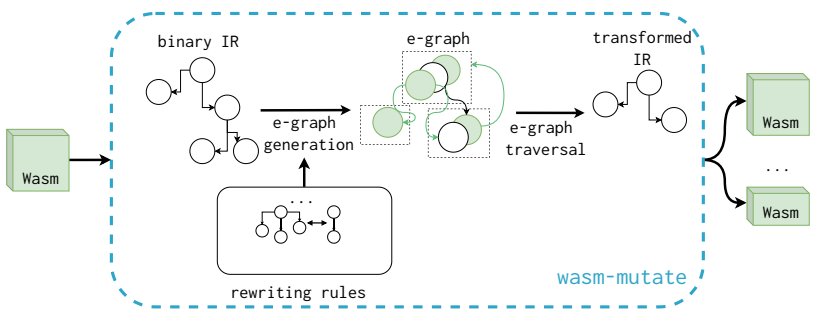

Figure 3.5: wasm-mutate high level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by a diversification space traversals using e-graphs(refer to 3.3).

[3] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

Figure 3.5 illustrates the workflow of wasm-mutate, which initiates with a WebAssembly binary as its input. The first step involves parsing this binary to create suitable abstractions, e.g., data flow graphs. Subsequently, wasm-mutate utilizes predefined rewriting rules to construct an e-graph for the initial program, encapsulating all potential equivalent codes derived from the rewriting rules. This stage exploits an essential e-graph property: each path traversed within the e-graph yields semantically equivalent code. Then, pieces of the original program are randomly substituted by the result of random e-graph traversals, resulting in a variant that maintains semantic equivalence to the original binary. This assurance of semantic preservation is rooted in the inherent properties of the individual rewrite rules employed. Notice that, the output of wasm-mutate can be used back as an input, facilitating the stacking of multiple transformations through repeated wasm-mutate iterations.

■ 3.3.1 WebAssembly Rewriting Rules

wasm-mutate incorporates a total of 135 rewriting rules, organized into categories referred to as meta-rules. A significant portion of these, 125 to be exact, fall under a peephole meta-rule. The rewriting rules are conceived based on the seminal work of Sasnauskas et al. [**?**], extended to include a predicate to enforce the conditions for replacement. Each rule is articulated as a tuple, represented as `(LHS, RHS, Cond)`, where: `LHS` identifies the segment of code targeted for replacement, `RHS` outlines the functionally equivalent substitute, and `Cond` defines the circumstances permitting the substitution. For example, in the case of WebAssembly binaries, the `Cond` predicate ensures that the replacement does not violate the type constraints of the `part`. The following text details seven prominent meta-rules utilized in wasm-mutate.

Add type: wasm-mutate implements two rewrite rules for the Type Section of input WebAssembly programs, one of which is illustrated in the following rewriting rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))

---

RHS (module
  (type (;0;) (func (param i32) (result i64)))
+ (type (;0;) (func (param i64) (result i32 i64)))
```

This transformation generates random function signatures with a random number of parameters and results count. This rewriting rule does not affect the runtime behavior of the variant. It also guarantees that the index of the already defined types is consistent after the addition of a new type.

Add function: The function and code sections of a Wasm binary contain function declarations and the code body of the declared functions, respectively. wasm-mutate add new functions, through mutations in the two mentioned sections. To add a new function, wasm-mutate creates a random type signature. Then, the random function body is created. The body of the function consists of returning the default value of the result type. wasm-mutate never adds a call instruction to this function. So in practice, the new function is never executed. The following example illustrates this rewriting rule.

```
LHS (module
   (type (;0;) (func (param i32 f32) (result i64)))

---

   RHS (module
      (type (;0;) (func (param i32 f32) (result i64)))
   +_______(func (;0;) (type 0) (param i32 f32) (result i64)
   +__________i64.const 0)
```

Remove dead code: wasm-mutate can randomly remove dead code. In particular wasm-mutate removes: *functions, types, custom sections, imports, tables, memories, globals, data segments and elements* that can be validated as dead code. For instance, to delete a memory declaration, the binary code must not contain a memory access operation. Separate mutators are included within wasm-mutate for each of the aforementioned elements. For a more concrete example, the following listing illustrates the case of a function removal.

```
LHS (module (type (func)))

---

RHS - (module (import "" "" (func)))

Cond The removed function is not called, it is not exported, and it is
     not in the binary _table.
```

When removing a function, wasm-mutate ensures that the resulting binary remains valid and semantically identical to the original binary: it checks that the deleted function was neither called within the binary code nor exported in the binary external interface. As exemplified above, wasm-mutate might also eliminate a function import while removing the function.

Edit custom sections: The custom section in WebAssembly is used to store metadata, such as the name of the compiler that produces the binary or the symbol information for debugging. Thus, this section does not affect the execution of the Wasm program. wasm-mutate includes one mutator to

edit custom sections. This is exemplified in the following rewriting rule.

```
LHS (module
...
    - (@custom "CS42" "zzz...")

-------------------------------------------------------------------------

RHS (module
...
    + (@custom "CS42" "xxx...")
```

The *Edit Custom Section* transformation operates by randomly modifying either the content or the name of the custom section.

If swapping: In WebAssembly, an if-construction consists of a consequence and an alternative. The branching condition is executed right before the `if` instruction; if the value at the top of the stack is greater than `0`, then the consequence-code is executed, otherwise the alternative-code is run. The *if swapping* rewriting swaps the consequence and alternative codes of an if-construction.

To swap an if-construction in WebAssembly, wasm-mutate inserts a negation of the value at the top of the stack right before the `if` instruction. In the following rewriting rule we show how wasm-mutate performs this rewriting.

```
LHS (module
    (func ...) (
condition C
        (if A else B end)
    )
)

-------------------------------------------------------------------------

RHS (module
    (func ...) (
condition C
i32.eqz
        (if B else A end)
    )
)
```

The consequence and alternative codes are annotated with the letters `A` and `B`, respectively. The condition of the if-construction is denoted as `C`. The negation of the condition is achieved by adding the `i32.eqz` instruction in the RHS part of the rewriting rule. The `i32.eqz` instruction compares the top value of the stack with zero, pushing the value `1` if the comparison is true. Some if-constructions may not have either a consequence or an alternative code. In such cases, wasm-mutate replaces the missing code block with a single `nop` instruction. In the context of ROP [**?**], this transformation can protect a victim binary to be exploited.

Loop Unrolling: Loop unrolling is a technique employed to enhance the performance of programs by reducing loop control overhead [**?**]. wasm-mutate incorporates a loop unrolling transformation and utilizes the Abstract Syntax Tree (AST) of the original Wasm binary to identify loop constructions.

When wasm-mutate selects a loop for unrolling, its instructions are divided by first-order breaks, which are jumps to the loop's start. This separation ensures that branching instructions controlling the loop body do not require label index adjustments during unrolling. The same holds true for instructions continuing to the next loop iteration. As the loop unrolling process unfolds, a new Wasm block is created to encompass both the duplicated loop body and the original loop. Within this newly established block, the previously separated groups of instructions are copied. These replicated groups of instructions mirror the original ones, except for branching instructions jumping outside the loop body, which need their jumping indices increased by one. This modification is required due to the introduction of a new `block ... end` scope around the loop body, which affects the scope levels of the branching instructions.

In the following text we illustrate the rewriting rule for a function that contains a loop.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
  )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'   br_if 0   B'   br 1   end)
            (loop   A'   br_if 0   B'   end)
        end) 8bb152
    )
)
```

The loop in the LHS part features a single first-order break, indicating that its execution will cause the program to continue iterating through the loop. The loop body concludes right before the `end` instruction, which highlights the point at which the original loop breaks and resumes program execution. Upon selecting the loop for unrolling, its instructions are divided into two groups, labeled `A` and `B`. As illustrated in the RHS part, the unrolling process entails creating two new Wasm blocks. The outer block encompasses both the original loop structure and the duplicated loop body, while the inner blocks, denoted as `A'` and `B'`, represent modifications of the jump instructions in groups `A` and `B`, respectively. Notice that, any jump instructions within `A'` and `B'` that originally leaped outside the loop must

have their jump indices incremented by one. This adjustment accounts for the new block scope introduced around the loop body during the unrolling process. Furthermore, an unconditional branch is placed at the end of the unrolled loop iteration's body. This ensures that if the loop body does not continue, the tool breaks out of the scope instead of proceeding to the non-unrolled loop.

Peephole: This transformation category is about rewriting instruction sequences within function bodies, signifying the most granular level of rewriting. We implement 125 rewriting rules for this group in wasm-mutate. We include rewriting rules that affects the memory of the binary. For example, we include rewriting rules that creates random assignments to newly created global variables. For these rules, we incorporate several conditions, denoted by `Cond`, to ensure successful replacement. These conditions can be utilized interchangeably and combined to constrain transformations (see 3.3).

For instance, wasm-mutate is designed to guarantee that instructions marked for replacement are deterministic. We specifically exclude instructions that could potentially cause undefined behavior, such as function calls, from being mutated. For this rewriting type, wasm-mutate only alters stack and memory operations, leaving the control frame labels unaffected.

■ 3.3.2 E-graphs traversal for variant generation

We build wasm-mutate on top of e-graphs [**?**]. An e-graph is a graph data structure utilized for representing rewriting rules and their chaining. In an e-graph, there are two types of nodes: e-nodes and e-classes. An e-node represents either an operator or an operand involved in the rewriting rule, while an e-class denotes the equivalence classes among e-nodes by grouping them, i.e., an e-class is a virtual node compound of a collection of e-nodes. Thus, e-classes contain at least one e-node. Edges within the graph establish operator-operand equivalence relations between e-nodes and e-classes.

In wasm-mutate, the e-graph is automatically built from a WebAssembly program by analyzing its expressions and operations through its data flow graph. Then, each unique expression, operator, and operand are transformed into e-nodes. Based on the input rewriting rules, the equivalent expressions are detected, grouping equivalent e-nodes into e-classes. During the detection of equivalent expressions, new operators could be added to the graph as e-nodes. Finally, e-nodes within an e-class are connected with edges to represent their equivalence relationships.

For example, let us consider one program with a single instruction that returns an integer constant, `i64.const 0`. Let us also assume a single rewriting rule, `(x, x i64.or x, x instanceof i64)`. In this example, the program's control flow graph contains just one node, representing

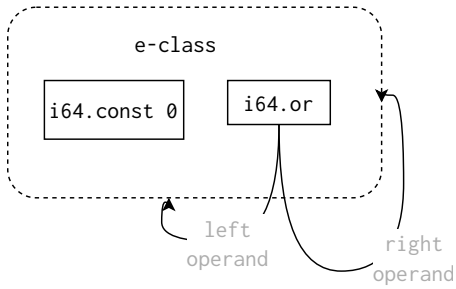

Figure 3.6: e-graph for idempotent bitwise-or rewriting rule. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

the unique instruction. The rewriting rule represents the equivalence for performing an `or` operation with two equal operands. Figure 3.6 displays the final e-graph data structure constructed out of this single program and rewriting rule. We start by adding the unique program instruction `i64.const 0` as an en e-node (depicted by the leftmost solid rectangle node in the figure). Next, we generate e-nodes from the rewriting rule (the rightmost solid rectangle) by introducing a new e-node, `i64.or`, and creating edges to the `x` e-node. Following this, we establish equivalence. The rewriting rule combines the two e-nodes into a single e-class (indicated by the dashed rectangle node in the figure). As a result, we update the edges to point to the `x` symbol e-class.

Willsey et al. illustrate that the extraction of code fragments from e-graphs can achieve a high level of flexibility, especially when the extraction process is recursively defined through a cost function applied to e-nodes and their operands. This approach guarantees the semantic equivalence of the extracted code [**?**]. For example, to obtain the smallest code from an e-graph, one could initiate the extraction process at an e-node and then choose the AST with the smallest size from among the operands of its associated e-class [**?**]. When the cost function is omitted from the extraction methodology, the following property emerges: *Any path traversed through the e-graph will result in a semantically equivalent code variant*. This concept is illustrated in Figure 3.6, where it is possible to construct an infinite sequence of "or" operations. In the current study, we leverage this inherent flexibility to generate mutated variants of an original program. The e-graph offers the option for random traversal, allowing for the random selection of an e-node within each e-class visited, thereby yielding an equivalent expression.

We propose and implement the following algorithm to randomly traverse an e-graph and generate semantically equivalent program variants, see 1. It receives an e-graph, an e-class node (initially the root's e-class), and the

Algorithm 1 e-graph traversal algorithm taken from [**?**].

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:
3:      if depth = 0 then
4:          return smallest_tree_from(egraph, eclass)
5:      else
6:          nodes ← egraph[eclass]
7:          node ← random_choice(nodes)
8:          expr ← (node, operands = [])
9:          for each child ∈ node.children do
10:             subexpr ← TRAVERSE(egraph, child, depth − 1)
11:             expr.operands ← expr.operands ∪ {subexpr}
12:         return expr
```

maximum depth of expression to extract. The depth parameter ensures that the algorithm is not stuck in an infinite recursion. We select a random e-node from the e-class (lines 5 and 6), and the process recursively continues with the children of the selected e-node (line 8) with a decreasing depth. As soon as the depth becomes zero, the algorithm returns the smallest expression out of the current e-class (line 3). The subexpressions are composed together (line 10) for each child, and then the entire expression is returned (line 11). To the best of our knowledge, wasm-mutate, is the first practical implementation of random e-graph traversal for WebAssembly.

Let's demonstrate how the proposed traversal algorithm can generate program variants with an example. We will illustrate Algorithm 1 using a maximum depth of 1. **??** presents a hypothetical original Wasm binary to mutate. In this example, the developer has established two rewriting rules: `(x, x i32.or x, x instanceof i32)` and `(x, x i32.add 0, x instanceof i32)`. The first rewriting rule represents the equivalence of performing an `or` operation with two equal operands, while the second rule signifies the equivalence of adding 0 to any numeric value. By employing the code and the rewriting rules, we can construct the e-graph depicted in Figure 3.7. The figure demonstrates the operator-operand relationship using arrows between the corresponding nodes.

In Figure 3.7, we annotate the various steps of Algorithm 1 for the scenario described above. Algorithm 1 begins at the e-class containing the single instruction `i64.const 1` from **??**. It then selects an equivalent node in the e-class (2), in this case, the `i64.or` node, resulting in: `expr = i64.or l r`. The traversal proceeds with the left operand of the selected node (3), choosing the `i64.add` node within the e-class: `expr = i64.or (i64.add l r) r`. The left operand of the `i64.add` node is the original node (5): `expr = i64.or (i64.add i64.const 1 r) r`. The right operand of the `i64.add` node belongs to another e-class, where the node `i64.const`

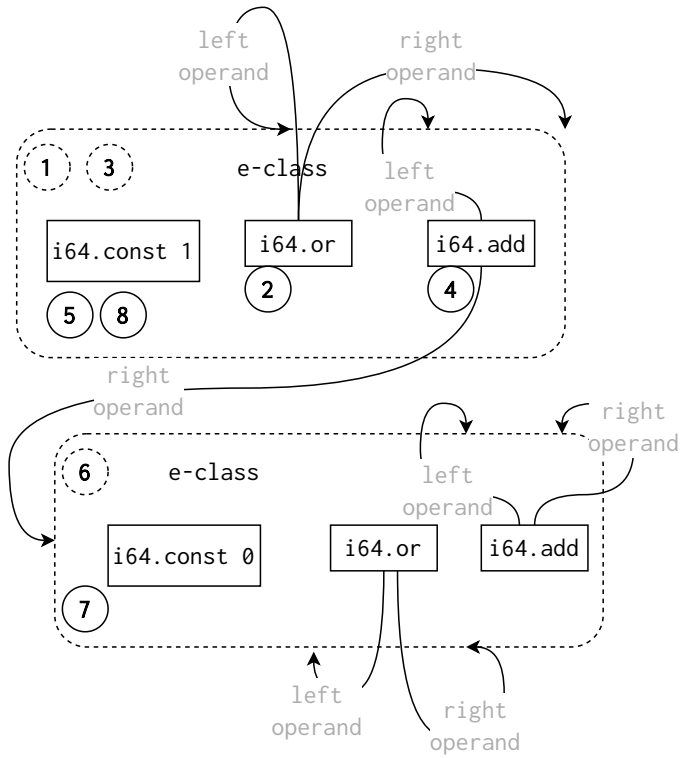

Figure 3.7: e-graph built for rewriting the first instruction of **??**.

`0` is selected (6)(7): `expr = i64.or (i64.add i64.const 1 i64.const 0) r`. In the final step (8), the right operand of the `i64.or` is selected, corresponding to the initial instruction e-node, returning: `expr = i64.or (i64.add i64.const 1 i64.const 0) i64.const 1` The traversal result applied to the original Wasm code can observed in **??**.

3.4 Discussion

3.4.1 Accomanying artifacts

wasm-mutate is implemented in Rust, comprising approximately, 10 thousands lines of Rust code. We leverage the capabilities of the wasm-tools project of the bytecodealliance for parsing and transforming WebAssembly binary code. Specifically, we utilize the wasmparser and wasm-encoder modules for parsing and encoding Wasm binaries, respectively. The implementation of wasm-mutate is publicly available for future research and can be found at .

TODO Comparison of the approaches. **TODO** Add the cool table.

TODO This is contradictory to our binary solution. We use the superdiversifier idea of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on handwritten rules . Concretely, we can control the quality of the generated codes. Besides, CROW always generates equivalent programs because it is based on a solver to check for equivalence. Second, there is a battle-tested superoptimizer for LLVM, Souper [**?**] This latter makes it feasible the construction of a generic LLVM superdiversifier.

04 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

4.1 Offensive Software Diversification

4.1.1 **Use case 1:** Automatic testing and fuzzing of WebAssembly consumers

TODO We explain the CVE. Make the explanation around "indirect memory diversification"

4.1.2 **Use case 2:** WebAssembly malware evasion

TODO The malware evasion paper

4.2 Defensive Software Diversification

4.2.1 **Use case 3:** Multivariant execution at the Edge

TODO Disturbing of execution time. Go around the web timing attacks.

4.2.2 **Use case 4:** Speculative Side-channel protection

In concrete, distributing the unmodified binary to 100 machines would, essentially, creates 100 homogeneously vulnerable machines. However, let us illustrate the case with a different approach: each time the binary is replicated onto a different machine, we distribute a unique variant instead of the original binary. If we disseminate a unique variant, with X stacked transformations, to each machine, every system would run a distinct Wasm binary. Based on our findings, even when some binaries are still vulnerable, we can confidently say that if 100 variants of a vulnerable program, each furnished with X stacked transformations, are distributed, the impact of any potential attack is considerably mitigated. While it's true that some

variants may retain their original vulnerabilities, not all of them do. This significantly enhances overall security. Further reinforcing this point, let's consider the case of btb_leakage. In this scenario, a suite of 100 variants, each featuring at least 200 stacked transformations, ensures full protection against potential threats, effectively securing the entire infrastructure. Moreover, considering the results for the ret2spec attack, this property holds for the whole population of generated variants, despite the number of stacked transformations. Therefore, wasm-mutate as a software diversification tool, is a peemptive solution to potential attacks.

TODO Go around the last paper

05 CONCLUSIONS AND FUTURE WORK

- 5.1 Summary of technical contributions
- 5.2 Summary of empirical findings
- 5.3 Summary of empirical findings
- 5.4 Future Work

[1] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," *Proceedings of the Web Conference 2021*, 2021.

[2] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[3] J. Cabrera Arteaga, P. Laperdrix, M. Monperrus, and B. Baudry, "Multi-Variant Execution at the Edge," *arXiv e-prints*, p. arXiv:2108.08125, Aug. 2021.

[4] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*, pp. 100–120, Springer, 2008.

[5] M. Henry, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Nov 1987.

[6] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[7] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, *et al.*, "Swivel: Hardening webassembly against spectre," in *USENIX Security Symposium*, 2021.

[8] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Sfi safety for native-compiled wasm," *NDSS. Internet Society*, 2021.

[9] "Webassembly system interface." `https://github.com/WebAssembly/WASI`, 2021.

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

https://doi.org/10.1145/3397537.3397567

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb

https://doi.org/10.14722/madweb.2021.23004

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)

https://dl.acm.org/doi/abs/10.1145/3560828.3564007

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

https://www.sciencedirect.com/science/article/pii/S0167404823002067

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)

https://doi.org/10.1145/3358504.3361228